AD-A254 050

URITY CLASSIFIC

MENTATION PAGE

(2)

DTIC ELECTE AUG 1 9 1992 S A D

REPORT SECURI	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited This document has been approved for public release and sale; its distribution is unlimited.
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

NAME OF PERFORMING ORGANIZATION Computer Sciences Department University of Wisconsin	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of the Chief of Naval Research
ADDRESS (City, State, and ZIP Code) 1210 West Dayton Street Madison, WI 53706		7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000
NAME OF FUNDING/SPONSORING ORGANIZATION DARPA	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ARPA Order No: 6378 ONR Contract No: N00014-88-K-0590
ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22209-2308		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

1. TITLE (Include Security Classification)
Final Report: Software Support for Programming in the Large

2. PERSONAL AUTHOR(S)
Reps, Thomas; Horwitz, Susan; Solomon, Marvin

3a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 7/88 TO 12/91	1992, August, 6	11

6. SUPPLEMENTARY NOTATION

7. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software development environment, version control, configuration management, program integration, program slicing, persistent object store

9. ABSTRACT (Continue on reverse if necessary and identify by block number)

The research supported under this contract concerned the design and implementation of interactive environments for computer programming. Activities were carried out in two main areas:

Semantics-based program integration

By *program integration* we mean the merging process that one has to go through when a program's source code diverges into multiple variants (*e.g.*, supporting different features, different operating systems, or incorporating different bug-fixes). The goal of our research is to create a system that tests whether the enhancements made to two or more variants of a program interfere, and – if there is no interference – automatically integrates (combines) the variants so as to incorporate all the different enhancements in *one* program. This would be applied, for example, when a number of collaborators are collectively producing updates in a large programming project.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas Reps	22b. TELEPHONE (Include Area Code) (608)262-2091	22c. OFFICE SYMBOL 5387

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

19. Abstract (Continued)

Logic-based tools for programming-in-the-large

The CAPITL program-development environment is an integrated collection to tools supporting cooperative development of moderate to large programs. It includes a database for storing software objects such as program source modules, executable programs, and documentation, as well as arbitrary properties and relationships among objects. The database efficiently supports multiple snapshots or *versions* of the objects and relationships. A sophisticated deductive query language – based on Prolog – supports configuration management.

Final Report: Software Support for Programming in the Large

(ARPA Order No.: 6378; ONR Contract No.: N00014-88-K-0590)

Thomas Reps (P.I.), Susan Horwitz, and Marvin Solomon
University of Wisconsin–Madison

August 6, 1992

Final Report: Software Support for Programming in the Large

(ARPA Order No.: 6378; ONR Contract No.: N00014-88-K-0590)

Thomas Reps (P.I.), Susan Horwitz, and Marvin Solomon
University of Wisconsin–Madison

1. Introduction

This report summarizes the activities that were carried out under ARPA Order No. 6378, monitored by the Office of Naval Research under contract N00014-88-K-0590. The contract provided full or partial support for three faculty members, two visiting faculty members, one post-doctoral associate, ten graduate students, one undergraduate student, and two staff programmers.[1]

The research supported by contract N00014-88-K-0590 concerns the design and implementation of interactive environments for computer programming; the goal is the development of powerful language-specific tools that support program development, debugging, and testing, and exploit state-of-the-art personal computing hardware. Activities were carried out in two main areas:

Semantics-based program integration

By *program integration* we mean the merging process that one has to go through when a program's source code diverges into multiple variants (*e.g.*, supporting different features, different operating systems, or incorporating different bug-fixes). The goal of our research is to create a system that tests whether the enhancements made to two or more variants of a program interfere, and—if there is no interference—automatically integrates (combines) the variants so as to incorporate all the different enhancements in *one* program. This would be applied, for example, when a number of collaborators are collectively producing updates in a large programming project.

Logic-based tools for programming-in-the-large

The CAPITL program-development environment is an integrated collection of tools supporting cooperative development of moderate to large programs. It includes a database for storing software objects such as program source modules, object modules, executable programs, and documentation, as well as arbitrary properties and relationships among objects. The database efficiently supports multiple snapshots or *versions* of the objects and relationships. A sophisticated deductive query language—based on Prolog—supports configuration management.

2. Semantics-Based Program Integration

Our goal is to design a *semantics-based* tool for program integration. We want a tool that—given program *Base* and two variants *A* and *B*—makes use of knowledge of the programming language to determine whether the changes made to *Base* to produce *A* and *B* have undesirable semantic interactions; only if there is no such interference should the tool produce a merged program *M*.

While our long-term goal is to design such a tool for a full-fledged programming language, for the short term we have been using a simplified model of the program-integration problem so as to make the problem amenable to theoretical study. This model possesses the essential features of the problem, and thus permits us to conduct our studies without being overwhelmed by inessential details. Our integration model has the following characteristics:

(1) We restrict our attention to the integration of programs written in a simplified programming language that has only assignment statements, conditional statements, while loops, and final output statements (called *end* statements); by definition, only those variables listed in the end statement have values in the final state. The language does not include input statements; however, a program can use a variable before assigning to it, in which case the variable's value comes from the initial state.

(2) When an integration algorithm is applied to base program *Base* and variant programs *A* and *B*, and if integration succeeds—producing program *M*—then for any initial state σ on which *Base*, *A*, and *B* all

[1]Other support for the work has been provided by the National Science Foundation under grants DCR-8552602, DCR-8603356, CCR-8958530, and CCR-9100424, by a David and Lucile Packard Fellowship for Science and Engineering, and by donations from Cray Research Foundation, DEC, Eastman Kodak, HP, IBM, Siemens, Xerox, and 3M.

terminate normally,[2] M must have the following properties:

(i) M terminates normally on σ.

(ii) For any variable x that has final value v after executing A on σ, and either no final value or a different final value v' after executing *Base* on σ, x has final value v after executing M on σ (*i.e.*, M agrees with A on x).

(iii) For any variable y that has final value v after executing B on σ, and either no final value or a different final value v' after executing *Base* on σ, y has final value v after executing M on σ (*i.e.*, M agrees with B on y).

(iv) For any variable z that has the same final value v after executing *Base*, A, and B on σ, z has final value v after executing M on σ (*i.e.*, M agrees with *Base*, A, and B on z).

(3) Program M is to be created only from components that occur in programs *Base*, A, and B.

A more informal statement of Property (2) is: changes in the behavior of A and B with respect to *Base* must be preserved in the integrated program, along with the unchanged behavior of all three.

Properties (1) and (3) are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. *Any* program M that satisfies Properties (1), (2), and (3) integrates *Base*, A, and B; if no such program exists then A and B interfere with respect to *Base*. However, Property (2) is not decidable, even under the restrictions given by Properties (1) and (3); consequently, any program-integration algorithm will sometimes report interference—and consequently fail to produce an integrated program—even though there is actually *no* interference (*i.e.*, even when there is *some* program that meets the criteria given above).

The Horwitz-Prins-Reps Algorithm for Program Integration

The first algorithm that meets the requirements given above was formulated by S. Horwitz, J. Prins, and T. Reps in early 1987 (see [Horwitz88, Reps88] and publication [6] in the list provided in Section 4). That algorithm—referred to hereafter as the *HPR algorithm*—is the first algorithm for semantics-based program integration.

The HPR algorithm represents a fundamental advance over text-based program-integration algorithms (such as the UNIX utility *diff3*), and provides the first step in the creation of a theoretical foundation for building a semantics-based program-integration tool. Changes in *behavior* rather than changes in text are detected, and are preserved in the integrated program. To be more precise, by the "behavior" of a program component on some initial state σ, we mean the sequence of values produced at the component when the program is executed on σ. By "the sequence of values produced at a component," we mean: for a predicate, the sequence of boolean values to which the predicate evaluates; for an assignment statement, the sequence of values assigned to the target variable; for the final use of a variable in an end statement, the singleton sequence containing the variable's final value.

Although it is undecidable to determine whether a program modification actually leads to a change in program behavior, it is possible to determine a safe approximation by comparing each of the variants with the original program *Base*. To determine this information, the HPR algorithm employs a program representation that is similar to the *program dependence graphs* (PDGs) that have been used previously in vectorizing and parallelizing compilers [Kuck81, Ferrante87]. The algorithm also makes use of Weiser's notion of a *program slice* [Weiser84, Ottenstein84] to find the statements of a program that determine the values of potentially affected variables.

Given PDGs G_{Base}, G_A, and G_B (for programs *Base*, A, and B, respectively), the HPR algorithm performs three steps. The first step identifies three subgraphs that represent the changed behavior of A with respect to *Base*, the changed behavior of B with respect to *Base*, and the behavior that is the same in all three programs. The second step combines these subgraphs to form a merged dependence graph G_M. The third step determines whether A and B interfere with respect to *Base*; if there is no interference, an integrated program M is produced from graph G_M.

Considerable effort was spent on correctness considerations related to the HPR algorithm. This work was reported in publications [16] and [26], which are summarized below.

[2]There are two ways in which a program may fail to terminate normally on some initial state: (1) the program contains a nonterminating loop, or (2) a fault occurs, such as division by zero.

Semantic foundations of the HPR algorithm (publication [16])

The HPR algorithm makes use of the notion of a program slice to find just those statements of a program that determine the values of potentially affected variables. Consequently, W. Yang and T. Reps studied the relationship between the execution behavior of a program and the execution behavior of its slices. Our main results were two theorems: one shows that a slice captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice; the other shows that if a program is decomposed into (two or more) slices, the program halts on any state for which all the slices halt. These results were then used to show that the HPR algorithm meets the semantic criterion for integration that was given at the beginning of this section as Property (2).

Correctness of an algorithm for reconstituting a program from a dependence graph (publication [26])

A rather involved argument was necessary to establish the correctness of the method used in the HPR algorithm for reconstituting the text of the integrated program from the merged program dependence graph.

A Prototype Program-Integration Tool

An important component of our work has been the implementation of a prototype program-integration system that realizes the HPR algorithm (see publications [32] and [27]). This work has been carried out by T. Bricker, G. Rosay, V. Barger, and T. Reps. The prototype integration system allows us to demonstrate the concepts of slicing and integration, and serves as a testbed for some of the ideas developed by members of the project that go beyond the HPR algorithm (see below). For example, it has been extended to demonstrate the algorithm we developed for efficiently testing whether two slices are isomorphic (see publication [2]).

The user interface for the integration tool incorporates a language-specific editor created using the Synthesizer Generator, a meta-system for creating interactive, language-specific program-development systems [Reps88a, Reps88b]. Data-flow analysis of programs is carried out according to the editor's defining attribute grammar and used to construct the underlying program dependence graphs. An integration command added to the editor invokes the integration algorithm on the program dependence graphs, reports whether the variant programs interfere, and, if there is no interference, builds the integrated program.

During Spring 1990 we began distributing the prototype integration system under license from the University of Wisconsin–Madison, and an enhanced version of the system was distributed in Spring 1992 (see publication [22]). The system has currently been licensed to eight sites.

Algebraic Properties of Program Integration

The issue here was to understand the algebraic properties of the program-integration operation, such as whether there are laws of associativity and distributivity. These are of interest when dealing with compositions of integrations. For example, if three variants of a given base program are to be integrated by a pair of (two-variant) integrations, it is important to know whether there is a law of associativity to guarantee that it does not matter which two variants are integrated first. (Such a law does, in fact, hold.)

To demonstrate such properties, we first reformulated the integration algorithm as an operation in a *Brouwerian algebra* constructed from sets of dependence-graph slices. (A Brouwerian algebra is a distributive lattice with an additional binary operation, denoted by $\dot{-}$, which is a kind of difference operation.) In this algebra, the program-integration operation can be defined solely in terms of $\sqcup$, $\sqcap$, and $\dot{-}$. By making use of the rich set of algebraic laws that hold in Brouwerian algebras, we were able to establish a number of the integration operation's algebraic properties (see publications [13] and [3]).

The prototype integration system was extended to implement these ideas by adding to the system an editor and an interpreter for a higher-order functional language that operates on values of type *Brouw*, where a Brouw value is a set of slices of a certain form. The primitive operations on Brouw values are the join, meet, and pseudo-difference of a Brouwerian lattice, together with a ternary operation for integration. Functional expressions are built up using lambda-abstraction, application, conditional expressions, let-clauses, and a least fixed-point operator. A free variable in an expression (say x) denotes the Brouw value created from the program in editing buffer x. If no such buffer exists, the value is $\perp$, the least element in the Brouwerian lattice of slice sets. An evaluation command added to the editor invokes the interpreter on the expression, and—if the final result is a Brouw value—builds the corresponding program (if one exists).

G. Ramalingam and T. Reps have also pursued some other approaches to studying the algebraic properties of program-integration algorithms (see publications [9], [8], and [24]).

Illustrating Interference

T. Bricker and T. Reps studied how an integration tool can illustrate the causes of interference to the user after interference is detected (see publication [14]). Our main technical result was an alternative characterization of the HPR algorithm's interference criterion that is more suitable for illustrating the causes of interference. We proposed a number of methods for an integration system to display information to demonstrate the causes of interference to the user. One of these methods was then incorporated into the prototype integration tool.

Identifying Syntactic and Semantic Differences in Versions of Programs

S. Horwitz and W. Yang studied techniques for identifying syntactic and semantic differences in two versions of a program. The techniques used by the HPR algorithm for identifying the changed behaviors of the variant programs with respect to the base program were adapted by Horwitz for use in a language-based tool for identifying the semantic and textual differences between two versions of a program (see publication [12]). The prototype integration system was extended to incorporate these ideas.

Yang's syntactic matching algorithm (see publication [4]) operates on the abstract-syntax tree representations of two programs. The algorithm finds a *maximum matching* between the two trees; after the matching, unmatched components are identified to the user as the syntactic differences between the two programs. An implementation of this algorithm (for C programs) was put in the public domain.

Extending the Range of Applicability of the HPR Algorithm

A major focus of our work has been on how to extend the set of language constructs to which our ideas about program-integration are applicable. These issues have been a fertile source of topics for Ph.D. dissertations.

Languages With Procedure Calls

D. Binkley, S. Horwitz, and T. Reps investigated definitions of integration and interference that are suitable for integration of programs in languages with procedure calls. We first considered the problem of interprocedural slicing—generating a slice of an entire program, where the slice crosses procedure boundaries. To solve this problem, we introduced a new kind of graph to represent programs, called a *system dependence graph*, which extends previous dependence representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs (see publication [30]). Our main result was an algorithm for interprocedural slicing that uses the new representation (see publications [17] and [5]).

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To handle this problem, system dependence graphs include some data-dependence edges that represent *transitive* dependences due to the effects of procedure calls, in addition to the conventional direct-dependence edges. These edges are constructed with the aid of an auxiliary structure that represents calling and parameter-linkage relationships. This structure takes the form of an attribute grammar. The step of computing the required transitive-dependence edges is reduced to the construction of the subordinate characteristic graphs for the grammar's nonterminals.

Results on the integration problem for programs in languages with procedure calls are presented in D. Binkley's Ph.D. dissertation (see publication [18]).

Accommodating Semantics-Preserving Transformations

A limitation of the HPR algorithm is that it is overly conservative in its definition of interference. For example, it will report interference (and hence fail to produce an integrated program) when one variant changes the way a computation is performed—without changing the values computed (*i.e.*, the change is a semantics-preserving transformation)—while the other variant adds code that uses the result of the computation. To address this limitation, W. Yang, S. Horwitz, and T. Reps devised a new integration algorithm—referred to hereafter as the *YHR algorithm*—that uses more powerful notions of equivalence than slice-equality and consequently is able to accommodate semantics-preserving transformations (see publications [29], [11], and [1]).

The YHR algorithm is parameterized by an auxiliary algorithm that determines, for each component of *Base*, *A*, and *B*, which other components are *congruent*. (Roughly, two components are congruent only if they compute the same sequences of values when their respective programs are executed on the same initial state.) The YHR algorithm then uses an operation called *limited slicing* to extract program fragments from the base program and its variants that are smaller than the fragments extracted by the HPR algorithm.

Because the YHR algorithm is parameterized by the congruence-testing algorithm used, the YHR algorithm is actually a *class* of integration algorithms that accommodate semantics-preserving transformations. Although in general it is undecidable to identify congruent vertices exactly, the YHR algorithm can employ any *safe* congruence-testing algorithm (*i.e.*, one that identifies a subset of the exact set of congruent pairs of vertices). Starting from an algorithm given by B. Alpern, M. Wegman, and K. Zadeck [Alpern88] we were able to develop one such congruence-testing algorithm (see publication [31]).

These results were the subject of W. Yang's Ph.D. dissertation (see publication [21]).

Languages With Pointer Variables

Past work by other researchers has provided techniques for determining data dependences for languages with scalar variables and arrays. Work carried out by P. Pfeiffer, S. Horwitz, and T. Reps focused on methods for determining data dependences for languages with *pointer-valued* variables and *heap-allocated* storage (*e.g.*, Lisp and Pascal). Using the framework of abstract interpretation, we defined a family of algorithms that compute safe approximations to (*i.e.*, supersets of) the flow, output, and anti-dependences of a program written in such a language. Our algorithms account for destructive updates to fields of a structure and thus are not limited to the cases where all structures are trees or acyclic graphs; they are applicable to programs that build cyclic structures. The abstract-interpretation framework allowed us to demonstrate the correctness of these algorithms (see publication [15]).

Techniques for creating dependence graphs for languages with pointer variables are presented in P. Pfeiffer's Ph.D. dissertation (see publication [19]).

3. Logic-Based Tools for Programming-in-the-Large

The CAPITL[3] program-development environment is an integrated collection of tools supporting cooperative development of moderate to large programs. The tools are all based on a persistent object store built with the Exodus database toolkit [Carey90]. Closely integrated with this object store are an interactive browsing interface constructed using the InterViews windowing system [Linton89], a Unix compatibility interface, and a deductive query language.

3.1. The CAPITL Object Store

CAPITL's persistent storage is structured as a labeled, directed graph of *terms*. A term is either an *atom* or an *internal node*. Atoms currently come in five flavors: integers, real numbers, printable strings, byte strings, and "variables." (Variables are explained later in the section on Congress.) An internal node has a character-string *label*, and a table of references to terms indexed by distinct strings called *selectors*. In other words, a CAPITL database may be thought of as a directed graph with leaves containing data and arcs and internal nodes labeled by character strings. If *t* is a term and *s* is one of its selectors, the *s attribute* of *t* (denoted *t.s*) is the term referenced by selector *s* in *t*.

CAPITL is "identity-based" in the sense that each term has a unique identity: References identify specific instances of terms. Two terms with identical contents may nonetheless be considered distinct. Terms are explicitly created, and creating a new term is different from changing the contents of an existing one. A term reference is similar to a pointer in Pascal or C except that term identities are never reused; a reference to a newly allocated term can never be confused with a reference to one created earlier. An internal node is similar to a C struct, Pascal record, SNOBOL table, AWK associative array, PostScript dictionary, or LISP atom. It differs from a struct or record in that the set of selectors may be changed dynamically and the value associated with a selector must be a non-nil pointer rather than data of arbitrary type.

CAPITL maintains multiple snapshots or *versions* of the database. Each operation accessing the CAPITL database is done in the context of a designated *current version*. Changes affect only this version. A new version is conceptually created by making a complete copy of the entire database. In fact, the implementation uses an algorithm due to Driscoll and others [Driscoll89] that only requires copies of the terms that have actually changed. The algorithm is also efficient in time: operations to switch versions, traverse a particular version of the graphs, or make changes are nearly as fast as they would be were multiple versions not supported.

Each version has a *version ID*, which is a non-empty sequence of positive integers. The root version is version "0". The ID of the first child of a given version is formed by incrementing the final component of

[3]Computer-Assisted Programming In The Large

the parent's id. Sibling versions are formed by appending zeros. For example, the children of version 1.3.2 would be labeled 1.3.3, 1.3.3.0, 1.3.3.0.0, *etc.* This numbering is similar to the scheme used by RCS and SCCS, and seems more natural than "Dewey decimal" numbering in the common case of long sequences of single-child versions. For example, a sequence of consecutive derivations from 1.3.2 would yield 1.3.3, 1.3.4, 1.3.5, *etc.* This numbering scheme can, however, become quite confusing when multiple versions are derived from the same parent. We expect that versions will normally be selected by symbolic name or other attributes stored in an index structure (itself stored in the database) rather than version ID. (This part of the database is still under development.)

Although an internal node may have any set of selectors, some terms are marked as *objects*. An object is a "heavier weight" internal node that is guaranteed to have certain selectors with "built-in" semantics. Objects are further classified as *directories*, *files*, *symbolic links*, and *other*. Some CAPITL operations (particularly those associated with the interactive browser) view the *extent* of an object to be the set of terms reachable from it by paths (sequences of selectors) that do not go through other objects.

Every object has a `contents` attribute. Directory, file, and link objects are collectively called *Unix-like* objects. Unix-like objects have integer attributes `owner`, `group`, `permissions`, `mtime`, `atime`, and `ctime`.

A directory object is similar to a Unix file-system directory. Its `contents` attribute is a list of Unix-like objects. Each object is contained in a unique parent directory. (CAPITL does not support the equivalent of Unix "hard" links.) Every object has a `directory` attribute that points to its parent (the root directory is its own parent), and a *name* attribute, which is a printable-string atom. The set of all Unix-like objects forms a tree. A *file* object corresponds to a Unix "plain" file. Its `contents` is a byte-string atom. A *symbolic link* object's `contents` is a printable-string atom. The distinction between file and link objects is only important in the context of the Exodus File System (EFS) described below.

3.2. Accessing a CAPITL database

A CAPITL database can be accessed and manipulated in (at least) four ways:

- Directly, through programs written in the E programming language, a persistent extension of C++ [Richardson]
- Through the Unix-compatible EFS interface.
- Through an interactive X-based browser.
- Through the *Congress* deductive query language.

CAPITL is written in the E programming language, so all of its structures can be accessed as data structures in E. For example, terms are all instances of the class `Term`, which exports such methods (member functions) as

```
boolean IsAtom();
```

which enquires whether the term is an atom, and

```
Term *Subterm(char *selector);
```

which returns the term referenced by a particular selector (if the term is *not* an atom). Class `Integer` is a subclass of `Term` with an `IntVal()` method that returns its integer value, and so on.

An interactive browsing interface has been written on top of the X window system using the InterViews toolkit.

3.3. EFS

The *Exodus File System* (EFS) allows a CAPITL database to be mounted and accessed as if it were a Unix filesystem. The EFS server daemon *efsd* listens on a UDP port for NFS service requests. The Unix *mount* command may be used to graft any subtree of a CAPITL database into an existing Unix directory tree. Once a CAPITL database is mounted, Unix programs can access Unix-like objects just as if they were actual Unix files, directories, and symbolic links. For example, the Unix `open` system call binds a file descriptor to a CAPITL object, and the Unix `read`, `write`, and `seek` system calls can access or modify its `contents` attribute (more precisely, they modify the value of the byte-string atom referenced by the `contents` selector). Because this facility is implemented by the NFS feature of the Unix kernel, neither client programs nor the Unix kernel need be modified in any way. EFS allows the other "Unix-defined" attributes (`owner`, `mtime`, *etc*) to be accessed through the `stat`, `lstat`, and `fstat` system calls and modified through Unix calls such as `chown`, `chmod`, and `utimes`. Other attributes are not accessible through the EFS interface.[4]

Path names have the same semantics in EFS as in Unix, although the implementation is different. Since hard links are not supported, each object lives in a unique directory and has a unique pathname. The final component of the pathname is stored in the object itself as its `name` attribute.

The EFS supports version selection though an extension of pathname syntax. A version ID followed by a colon is interpreted as a request to resolve a pathname in a designated version of the database. Pathnames without version ID's are resolved in the current default version. For example,

```
        diff 3.3:prog.c prog.c
```

compares version 3.3 of `prog.c` with the current version, and

```
        (echo -n "updates done "; date) >> 3.5:log
```

adds a line to version 3.5 of `log`. A version ID can actually appear anywhere in a path name, although complicated pathnames with embedded version ID's may produce surprising results. The notation *ID*: at the end of a pathname or followed by "/" is an abbreviation for "*ID*:.". As in Unix, a pathname that does not start with "/" is interpreted relative to the current directory (and version).

Since the Unix kernel uses the same mechanism to resolve `chdir` requests as `open`, the shell's `cd` command can be used to navigate among versions. For example,

```
        cd 3.2.1:
```

sets 3.2.1 as the default version for subsequent file-system requests. The pathname supplied in a *mount* request is interpreted in the same way, so a default version can be specified at mount time, as in

```
        mkdir project.old
        mkdir project.new
        mount capitl:/3.4: project.old
        mount capitl:/3.5: project.new.
        cd project.new/include
        vi defs.h
```

3.4. Congress

Depending on your point of view, Congress may be viewed as a logic programming language, a deductive database query language, an embedded query language, or a library of classes for convenient database access. Since Congress is implemented as a library of classes, any E program can use Congress as a "higher level" alternative or enhancement to the raw E interface. Congress programs not only manipulate CAPITL terms, they *are* CAPITL terms, so they can be stored in the database and manipulated and invoked from the interactive interface. Congress is an embedded query language for E. A Congress query can be invoked from within an E program; the result is (loosely speaking) a set of `Term` data structures that can be processed by the E program. Congress also provides a convenient way for E procedures to be invoked by Congress programs, so in a sense, E is also embedded in Congress. Congress also has a character-string syntax, so that *ad hoc* queries or whole programs can be typed in from the keyboard.

Congress is closely modeled on the LOGIN language of Ait-Kaci [Ait-Kaci86] which in turn is derived from Prolog [Clocksin84]. The main differences between LOGIN and Prolog (for our purposes) are that subterms are identified by selectors rather than position, and that terms are not restricted to trees, but can be arbitrary (cyclic) graphs. The main extensions to LOGIN provided by Congress are a concept of object identity and an assignment operation that allows terms to be updated in place, extending the *assert* and *retract* operations in standard Prolog.

A Congress *program* is a set of *procedures*, each of which is a list of *clauses*. A clause consists of a term called the *head* of the clause, and a sequence of zero or more terms called the *body*. A clause with an empty body is called a *fact*. Since Congress programs are built out of terms, they can be stored (in "parsed" form) in the object store. More importantly, all terms in the object store can be treated as facts by a Congress program.

A recent Ph.D. thesis (publication [20) explores the application of logic programming to the mega-programming task of configuration management. Each object is represented by a term that describes the properties of the object in great detail. The descriptive information has the structure of a *type* in a rich higher-order polymorphic type calculus. These descriptions are provided for source objects (program

[4]Extensions to support EFS access to other attributes are under consideration.

source modules) as well as *tools*: executable programs that transform and combine other objects. Each type has two components, a *form* and a *functionality*. For example, the form of a Pascal compiler written in C is "C-source"; its functionality is to translate an object whose form is "Pascal-source" into an object of form "object code" while preserving functionality.

A request for a derived object is presented in the form of a *goal* term. The *contents* attribute of the goal is left unspecified. Other attributes may be unspecified or partially specified. A *planner* program (written in Congress) figures out how to build an object matching the goal by applying existing or derived tools to existing or derived objects. In effect, given a type, the planner finds an expression of that type. Sophisticated inferences are possible. For example, given an application program written in Pascal, a Pascal compiler written in C, and an executable C compiler, the planner will construct a plan to build an executable Pascal compiler and use it to compile the application.

4. Publications Citing Grant N00014-88-K-0590

Journal Publications

[1] Yang, W., Horwitz, S., and Reps, T., "A program integration algorithm that accommodates semantics-preserving transformations," To appear in *ACM Transactions on Software Engineering and Methodology*.

[2] Horwitz, S. and Reps, T., "Efficient comparison of program slices," *Acta Informatica 28* (1991), 713-732.

[3] Reps, T., "Algebraic properties of program integration," *Science of Computer Programming 17* (1991), 139-215.

[4] Yang, W., "Identifying syntactic differences between two programs," *Software – Practice & Experience* 21, 7 (July 1991), 739-755.

[5] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems 12*, 1 (January 1990), 26-60.

[6] Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345-387.

Invited Papers

[7] Horwitz, S. and Reps, T., "The use of program dependence graphs in software engineering," In *Proceedings of the Fourteenth International Conference on Software Engineering*, (May 11-15, 1992, Melbourne, Australia), ACM, New York, NY, 1992.

Conference Publications

[8] Ramalingam, G., and Reps, T., "Modification algebras," to appear in *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology (AMAST)*, (Iowa City, Iowa, May 22-24, 1991).

[9] Ramalingam, G. and Reps, T., "A theory of program modifications," in *Proceedings of the Colloquium on Combining Paradigms for Software Development*, (Brighton, UK, April 8-12, 1991), *Lecture Notes in Computer Science*, Vol. 494, S. Abramsky and T.S.E. Maibaum (eds.), Springer-Verlag, New York, NY, 1991, pp. 137-152.

[10] Rich, A. and Solomon, M., "A logic-based approach to system modelling," in *Proceedings of the Third International Workshop on Software Configuration Management*, (Trondheim, Norway, June 1991).

[11] Yang, W., Horwitz, S., and Reps, T., "A program integration algorithm that accommodates semantics-preserving transformations," in *SIGSOFT '90: Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, (Irvine, CA, December 3-5, 1990), pp. 133-143. Appeared as: *ACM Software Engineering Notes 15*, 6 (December 1990).

[12] Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, (White Plains, NY, June 20-22, 1990), pp. 234-245. Appeared as: *ACM SIGPLAN Notices 25*, 6 (June 1990).

[13] Reps, T., "Algebraic properties of program integration," in *Proceedings of the 3nd European Symposium on Programming* (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science*, Vol. 432, N. Jones (ed.), Springer-Verlag, New York, NY, 1990, pp. 326-340.
Invited for a special issue of *Science of Computer Programming* (see [3]).

[14] Reps, T. and Bricker, T., "Illustrating interference in interfering versions of programs," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, (Princeton, NJ, October 24-27, 1989), pp. 46-55. Appeared as: *ACM Software Engineering Notes 17*, 7 (November 1989).

[15] Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," in *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), pp. 28-40. Appeared as: *ACM SIGPLAN Notices 24*, 7 (July 1989).

[16] Reps, T. and Yang, W., "The semantics of program slicing and program integration," in *Proceedings of the Colloquium on Current Issues in Programming Languages*, (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Vol. 352, J. Diaz and F. Orejas (eds.), Springer-Verlag, New York, NY, 1989, pp. 360-374.

[17] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), pp. 35-46. Appeared as: *ACM SIGPLAN Notices 23*, 7 (July 1988).

Ph.D. Dissertations

[18] Binkley, D. Multi-procedure program integration, Ph.D. dissertation and Tech. Rep. TR-1038, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1991.

[19] Pfeiffer, P. Dependence-based representations for programs with reference variables, Ph.D. dissertation and Tech. Rep. TR-1037, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1991.

[20] Rich, A. Logic-based system modelling, Ph.D. dissertation, Computer Sciences Department, University of Wisconsin, Madison, WI, May 1991.

[21] Yang, W. "A new algorithm for semantics-based program integration," Ph.D. dissertation and Tech. Rep. TR-962, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1990.

Software

[22] Reps, T., Bricker, T., *et al.*, *The Wisconsin Program-Integration System*. Release 0.5, April 1990. Release 1.0, April 1992. Licensed to 8 sites.

Other Publications and Reports

[23] Ball, T. and Horwitz, S. "Constructing control flow from control dependence," TR-1091, Computer Sciences Department, University of Wisconsin—Madison, June 1992.

[24] Ramalingam, G. and Reps, T., "New programs from old," TR-1057, Computer Sciences Department, University of Wisconsin–Madison, November 1991.
Submitted for journal publication.

[25] Ramalingam, G. and Reps, T., "On the computational complexity of incremental algorithms," TR-1033, Computer Sciences Department, University of Wisconsin–Madison, August 1991.
Submitted for conference and journal publication.

[26] Ball, T., Horwitz, S., and Reps, T., "Correctness of an algorithm for reconstituting a program from a dependence graph," TR-947, Computer Sciences Department, University of Wisconsin–Madison, July 1990.

[27] Reps, T., *The Wisconsin Program-Integration System Reference Manual*. Computer Sciences Department, University of Wisconsin–Madison, April 1990.

[28] Ramalingam, G. and Reps, T., "Semantics of program representation graphs," TR-900, Computer Sciences Department, University of Wisconsin–Madison, December 1989.

[29] Yang, W., Horwitz, S., and Reps, T., "A new program integration algorithm," TR-899, Computer Sciences Department, University of Wisconsin–Madison, December 1989.

[30] Binkley, D., Horwitz, S., and Reps, T., "The multi-procedure equivalence theorem," TR-890, Computer Sciences Department, University of Wisconsin–Madison, November 1989.

[31] Yang, W., Horwitz, S., and Reps, T., "Detecting program components with equivalent behaviors," TR-840, Computer Sciences Department, University of Wisconsin–Madison, April 1989.

[32] Reps, T. "Demonstration of a prototype tool for program integration," TR-819, Computer Sciences Department, University of Wisconsin–Madison, January 1989.

5. Additional References

Ait-Kaci86.
Ait-Kaci, H. and Nasr, R., "LOGIN: A logic programming language with built-in inheritance," *Journal of Logic Programming*, pp. 181-215 (March 1986).

Alpern88.
Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Carey90.
Carey, M., DeWitt, D., Graefe, G., Haight, D., Richardson, J., Schuh, D., Shekita, E., and Vandenberg, S., "The

EXODUS extensible DBMS project: An overview," in *Readings in Object-Oriented Databases*, ed. D. Maier,Morgan-Kaufman (1990).

Clocksin84.
Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, New York, NY (1984).

Driscoll89.
Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E., "Making data structures persistent," *Journal of Computer and System Sciences* 38(1) pp. 86-124 (February 1989).

Ferrante87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* 9(3) pp. 319-349 (July 1987).

Horwitz88.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Kuck81.
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

Linton89.
Linton, M.A., Vlissides, J.M., and Calder, P.R., "Composing user interfaces with InterViews," *IEEE Computer*, pp. 8-24 (February 1989).

Ottenstein84.
Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* 19(5) pp. 177-184 (May 1984).

Reps88.
Reps, T. and Horwitz, S., "Semantics-based program integration," pp. 1-20 in *Proceedings of the Second European Symposium on Programming*, (Nancy, France, March 21-24, 1988), *Lecture Notes in Computer Science*, Vol. 300, ed. H. Ganzinger,Springer-Verlag, New York, NY (1988).

Reps88a.
Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

Reps88b.
Reps, T. and Teitelbaum, T., *The Synthesizer Generator Reference Manual: Third Edition*, Springer-Verlag, New York, NY (1988).

Richardson.
Richardson, J., Carey, M., and Schuh, D., "The design of the E programming language," *ACM Trans. Program. Lang. Syst.*, (). To appear.

Weiser84.
Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* SE-10(4) pp. 352-357 (July 1984).